Exploring AArch64 assembler – Chapter 4

Oct 23, 2016 • Roger Ferrer Ibáñez • AArch64

In this chapter we will see some instructions that will allow us to compute things.

Arithmetic instructions

Since a computer is nothing but a glorified calculator (or a calculator it is nothing but a tiny computer), it has to be possible to perform basic arithmetic. For now we will restrict to integer operations. In later chapters we will see how we can manipulate other kinds of numbers.

Addition and subtraction

We can perform addition and subtraction using `add` and `sub` instructions. These instructions are pretty flexible in that they allow using many forms.

```
add Rdest, Rsource1, #immediate          // Rdest ← Rsource1 + immediate
add Rdest, Rsource1, Rsource2            // Rdest ← Rsource1 + Rsource2
add Xdest, Xsource1, Xsource2, shiftop   // Xdest ← Xsource1 + shiftop(Xsource2)
add Xdest, Xsource1, Xsource2, extop     // Xdest ← Xsource1 + extop(Xsource2)
add Wdest, Wsource1, Wsource2, shiftop   // Wdest ← Wsource1 + shiftop(Wsource2)
add Wdest, Wsource1, Wsource2, extop     // Wdest ← Wsource1 + extop(Wsource2)
add Xdest, Xsource1, Wsource2, extop     // Xdest ← Xsource1 + extop(Wsource2)
```

In the forms above `Rx` means either `Xx` or `Wx` (but without mixing them in the same instruction), `shiftop` and `extop` are the shift and extension operands described in chapter 3. In this case, `shiftop` does not include `ROR`. All the forms shown for `add` can be used for `sub` as well.

Multiplication and division

Compared to addition and subtraction, multiplication and division are harder operations. And there are a few different instructions for this purpose.

Due to the nature of multiplication, multiplying two values of 32/64 bits may end requiring 64/128 bits to be able to fully encode the mathematical result. If we know this will not happen (i.e. the result of

the value can be encoded in 32/64 bits) or we do not care we can use the `mul` instruction. If there are excess bits, they will be dropped.

```
mul Rdest, Rsource1, Rsource2 // Rdest ← Rsource1 * Rsource2
                              // but be careful with overflow
```

If we do care about the excess bits, then we have a bunch of instructions we can use. For 32-bit multiplications, `umull` and `smull` can be used. The latter is required when multiplying numbers in two's complement so the sign bit is correctly handled.

```
umull Xdest, Wsource1, Wsource2  // Xdest ← Wsource1 * Wsource2
smull Xdest, Wsource1, Wsource2  // Xdest ← Wsource1 * Wsource2
                                 // for two's complement numbers
                                 // in Wsource1 and Wsource2
```

For the less common case where we multiply two 64-bit registers and still we care about the upper 64-bits of the 128-bit result, then we can use `umulh` and `smulh`. For this case we will need two 64-bit registers (named `Xlower` and `Xupper` in the example below).

```
mul Xlower, Xsource1, Xsource2   // Xlower ← Lower64Bits(Xsource1 * Xsource2)
smulh Xupper, Xsource1, Xsource2 // Xupper ← Upper64Bits(Xsource1 * Xsource2)
```

Division is a bit simpler as only two instructions are necessary: `udiv` and `sdiv`. Again, the latter is for integer numbers encoded in two's complement.

```
udiv Rdest, Rsource1, Rsource2  // Rdest ← Rsource1 / Rsource2
sdiv Rdest, Rsource1, Rsource2  // Rdest ← Rsource1 / Rsource2
                                // when Rsource1, Rsource2 are
                                // in two's complement
```

These two instructions will compute the quotient of the division rounded towards zero.

Bitwise instructions

Bitwise instructions directly operate in the bits of the registers, without assuming any encoding in them.

Instruction `mvn` performs a bitwise not on its operand.

```
mvn Rdest, Rsource    // Rdest ← ~Rsource
```

Most of the bitwise instructions use two source registers. The basic ones are `and`, `orr` and `eor` (exclusive orr). They perform a bitwise and, a bitwise or and a bitwise xor respectively.

```
and Rdest, Rsource1, #immediate          // Rdest ← Rsource1 & immediate
and Rdest, Rsource1, Rsource2            // Rdest ← Rsource1 & Rsource2
and Xdest, Xsource1, Xsource2, shiftop   // Xdest ← Xsource1 & shiftop(Xsource2)
and Wdest, Wsource1, Wsource2, shiftop   // Wdest ← Wsource1 & shiftop(Wsource2)
```

Similar forms as the shown above exist for `orr` and `eor`. For bitwise instructions `shiftop` **does** include `ROR`.

There are combined versions of `mvn` plus `and`, `orr` and `eor` called `bic` (bit clear), `orn` (or not) and `eon` (exclusive or not) respectively. In this case, the second operand is first applied a NOT operation. They have slightly more limited forms.

```
orn Rdest, Rsource1, Rsource2            // Rdest ← Rsource1 | ~Rsource2
orn Xdest, Xsource1, Xsource2, shiftop   // Xdest ← Xsource1 | ~shiftop(Xsource2)
orn Wdest, Wsource1, Wsource2, shiftop   // Wdest ← Wsource1 | ~shiftop(Wsource2)
```

Likewise for `bic` and `eon`.

There are more instructions with narrower use cases, so we will omit them for now.

This is all for today.

« Exploring AArch64 assembler – Chapter 3 ARM assembler in Raspberry Pi – Chapter 26 »

Powered by Jekyll. Theme based on whiteglass
Subscribe via RSS